Journal of Parallel and Distributed Computing 151 (2021) 86–93

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

# Distributed programming of a hyperspectral image registration algorithm for heterogeneous GPU clusters

Jorge Fernández-Fabeiro, Arturo Gonzalez-Escribano, Diego R. Llanos *

*Departamento de Informática, Universidad de Valladolid, Spain*

ARTICLE INFO

*Article history:*
Received 24 April 2020
Received in revised form 29 January 2021
Accepted 8 February 2021
Available online 17 February 2021

*Keywords:*
Hyperspectral imaging
Image registration
Heterogeneous computing
Distributed arrays
Load balancing

ABSTRACT

Hyperspectral image registration is a relevant task for real-time applications such as environmental disaster management or search and rescue scenarios. The HYFMGPU algorithm was proposed as a single-GPU high-performance solution, but the need for a distributed version has arisen due to the continuous evolution of sensors that generate images with finer spatial and spectral resolutions. In a previous work, we simplified the programming of the multi-device parts of an initial MPI+CUDA multi-GPU implementation of HYFMGPU by means of Hitmap, a library to ease the programming of parallel applications based on distributed arrays. The performance of that Hitmap version was assessed in a homogeneous GPU cluster. In this paper, we extend this implementation by means of new functionalities added to the latest version of Hitmap in order to support arbitrary load distributions for multi-node heterogeneous GPU clusters. Three different load balancing layouts are tested, which prove that selecting a proper layout affects the performance of the code and how this performance is correlated with the use of the GPUs available in the cluster.



## 1. Introduction

Image registration is the task of estimating the translation, rotation and scaling parameters of a given image with respect to a second take of the same scene, obtained at different times, viewpoints, and/or lighting conditions. During the last few years, different hyperspectral image registration techniques have been proposed, but most do not focus on time performance. However, many real-time applications, such as the management of natural disasters or surveillance operations, depend on hyperspectral images being processed in real-time. GPUs were used to boost classification, target detection or segmentation of this kind of images, but few efforts were made to achieve a real-time implementation of a hyperspectral registration algorithm. Ordóñez et al. introduced in [25] a sequential CPU implementation of HYFM [29], a Fourier–Mellin algorithm for hyperspectral image registration. That work was followed by HYFMGPU, a single-GPU CUDA-based version whose performance makes it suitable for use in real-time environments [26]. As hyperspectral sensor technology improves, images have finer resolutions in both spatial and spectral domains. Because of that, more computational power and more memory space, this latter being a limited resource in GPUs, are needed. We presented in [12] a coarse-grained distributed multi-GPU implementation of HYFM that follows a hybrid MPI+CUDA programming approach which was proven to satisfy such present and future needs. That work also revealed the programming complexities of MPI when combined with an accelerator programming model such as CUDA, mixing work redistributions with data movements between host and devices, and calls to external scientific libraries. As a solution to those issues, a more abstract programming approach was introduced in [11]. This approach is based on Hitmap [17], a library designed to ease the task of programming parallel applications by means of its capabilities of automatic partitioning and mapping of distributed arrays with arbitrary granularity. It was proven to achieve not only a reduction in the overall programming complexity of the multi-GPU implementation of HYFM, but also an unexpected speed-up in the wall time of the algorithm when tested in a single-node homogeneous multi-GPU cluster. This homogeneity allows an easy balancing of the workload by processing the same amount of data on each accelerator [6], but the need for a more sophisticated load balancing mechanism is expected to arise when distributing the work among different kinds of accelerators in a heterogeneous GPU cluster.

In this paper, we present an extension of our previous HYFM multi-GPU implementation, introducing a mechanism that adapts the workload to the GPUs available in a cluster, improving its performance. This mechanism is supported by the new weight-distributed arrays included in the latest versions of Hitmap [5].[1]

* Corresponding author.
*E-mail addresses:* jorge@infor.uva.es (J. Fernández-Fabeiro), arturo@infor.uva.es (A. Gonzalez-Escribano), diego@infor.uva.es (D.R. Llanos).

[1] https://trasgo.infor.uva.es/hitmap/.

https://doi.org/10.1016/j.jpdc.2021.02.014
0743-7315/

An experimental study to assess the impact of a proper load balancing in terms of performance has been conducted on a cluster composed of two nodes that host five NVIDIA GPUs of different capabilities and generations. Five pairs of hyperspectral images of different sizes have been registered, distributing the work according to three different data layouts. By means of profiling the usage of each device, it has also been checked to what extent each layout is really balanced or not in terms of GPU time. The results reveal that the fastest option tested is a layout that distributes the load according to the performance differences observed when executing a single and small test case on each different GPU family. This fastest layout option presents a good and stable load balance since its standard deviation is barely 6.25% over the average GPU time. The results also show that the bigger the inputs are, the larger the impact that a proper load balancing layout has in the performance of this code.

The rest of this paper is organized as follows: we start by discussing some related research in Section 2 and providing the reader with an overview of the Hitmap distributed programming library in Section 3. Section 4 recalls the Hitmap-based multi-GPU implementation of the HYFM algorithm and introduces how the load balancing techniques of the library are applied to extend it. The results obtained by this extended approach are introduced in Section 5, and finally Section 6 presents the conclusions and some feasible research lines for the future.

## 2. Related work

GPU offloading is quite a common approach used in remote sensing to boost the implementation of the algorithms. For example, GPU-accelerated methods for geospatial change detection for both civilian [18,20] and military [34] fields have recently been presented, as well as for undersea image reconstruction [31] or synthetic aperture radar imaging in marine surfaces [19]. Many of these works also remark the sustained improvement of sensor technology and how it increases both computing and memory needs of any algorithm that manipulates data obtained in real-time from sensors embedded in unmanned vehicles or satellites. There are interesting research lines focused on exploiting heterogeneous devices in order to accelerate the relevant stages of such tasks. For instance, Martel et al. introduce in [22] some strategies to reach latency-efficient implementations of dimensionality reduction algorithms in a GPU and a Field Programmable Gate Array (FPGA). FPGAs are also used to accelerate such tasks as real-time target and anomaly detection [32] or image reconstruction from incomplete data [33]. Other works exploit highly-optimized GPU-based deep learning techniques to implement such remote sensing tasks as image-based crop health monitoring [28], cloud segmentation for weather prediction [10], or urban land segmentation and classification [16].

Algorithms that benefit from GPU offloading require an unreasonably long compute time on a single device, or are too large to fit into its memory. In such cases, work scattering among several GPUs is a common solution to overcome these limitations. However, depending on the properties of the algorithm being distributed, this approach may also require implementing complex domain decompositions [7,15] based on native programming interfaces such as MPI or CUDA.

Hitmap [17] is a compiler-agnostic library with a plain-C language API that offers an intermediate abstraction layer to tackle many of these issues, halfway between the manual programming of distributed data structures on message-passing models, and PGAS languages (Partitioned Global Address Space), such as Chapel [8] or UPC [21]. It provides a simple way to create distributed arrays that map to local address spaces, with explicit mechanisms for the construction of reusable communication patterns at runtime. These patterns adapt to the data partition, creating a low number of aggregated communications when moving data across the global space. This leads, for example, to a performance efficiency comparable to UPC, with a reduced programming complexity and development effort. Hitmap extends and generalizes the memory-hierarchy creation and data-partition functionalities of other libraries or distributed arrays models, such as HTAs [13] or Parray [9]. It allows the use of transparent partition policies, either regular or irregular, defined as interchangeable modules with a common interface. This hides for the programmer the decisions about granularity and synchronization across hierarchical levels. Hitmap has also been extended to support data structures such as sparse matrices, or graphs, using the same methodology and interface [14]. Hitmap is also the portable library used to provide a common interface for transparent data management in the Controller model [24], an abstract entity that allows programmers to easily manage the communications and kernel launching details, on multiple heterogeneous devices, including GPUs and multi-core CPUs [23]. Regarding its application in specific use cases, Hitmap has been presented in [30] as a programming interface to transparently map agents to processes in a multi-agent pedestrian simulator.

## 3. Overview of the Hitmap library

Since this work is an extension of the Hitmap-based multi-GPU implementation of the HYFM algorithm presented in [11], let us provide the reader with an overview of that distributed programming library. Hitmap [17] is a library for the partition, mapping, and management of hierarchically distributed data structures at runtime. It was originally designed for dense arrays, and has also been extended to support sparse data structures, such as sparse matrices or graphs, using the same methodology and interface [14]. It is based on an SPMD (Single Program Multiple Data) model and the message-passing paradigm. Hitmap defines several abstractions to write parallel programs using distributed data structures. The functions in the library are grouped in three main modules.

*Tiling functions.* They allow the definition and management of hierarchically tiled data structures. These functionalities can be used independently of the rest of the library to improve locality on sequential code. They define classes to represent domains of indexes in a compact form. A class named `HitTile` represents the association between the elements of the indexes-domain space and the actual data, allowing the access to data with the same efficiency as manually developed codes that do not use the tile abstraction. A process can declare and allocate a subspace of the original domain to create a distributed data structure.

*Mapping functions.* These include interchangeable modules that implement policies to automatically part and map domains in terms of the processes of a virtual topology. The virtual topologies are also generated by another class of policy modules at runtime. Neighbor relations across processes are established by these policies. The partitions are represented by objects named `HitLayout` that can be queried to obtain the indexes subdomain mapped to the local, a neighbor, or any other remote virtual process. Along with other partitions such as block, cyclic, or dimension-based, which were already supported in Hitmap, a number of weighted partitions [5] have been added to the latest versions of the library. These new partitions allow programmers to balance the workload among the processes in a topology, in order to match the capabilities of each computing device in a distributed heterogeneous system. Moreover, new distributed array constructors to directly map and allocate distributed arrays in terms of the selected topology and layout function names have also been added.

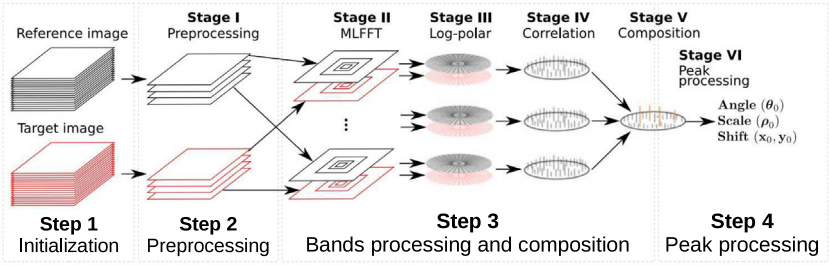


**Fig. 1.** HYFM scheme for registration of two hyperspectral images.
*Source:* Adapted from [26].

*Communication functions.* These are an abstraction of the message-passing model for tiles or tile parts across virtual processes. They allow the creation of `HitCom` objects that store the information needed to marshall/unmarshall and exchange selected tile data across processes. Several interfaces for different types of point-to-point and collective communications are available. More complex patterns composed of multiple communication operations, involving one or more tiles (several `HitCom` objects), are implemented as `HitPattern` objects. The constructor functions use the `HitLayout` objects associated to the distributed arrays to automatically determine what to communicate to which processes. Thus, these objects are transparently adapted on construction time to the target platform details and the actual data distribution selected. These communication objects have a method that can be called at any time, and as many times as needed, to execute the communications. Internally, these objects exploit efficient MPI techniques such as derived datatypes, asynchronous communications, etc.

## 4. Hitmap-based distributed HYFMGPU

The HYFM algorithm expects a pair of hyperspectral images (*reference* and *target*) as inputs. The goal is to register the target image. This means computing how it is rotated, shifted and scaled with respect to the reference image. This procedure was first implemented in CUDA by Ordóñez el al. to be run in single NVIDIA GPUs [26] and then distributed among several devices, as we presented in [12]. In this section, we offer an overview of the coarse-grain parallelization process followed to reach the distributed version of the algorithm implemented with Hitmap and presented in [11], describing also the modifications needed to adapt that code to a heterogeneous GPU cluster. In order to help the reader to gain a better understanding of the computation and communication phases of the algorithm, the rest of the section is organized according to the four main steps depicted in Fig. 1.

### 4.1. Initialization

Both the reference and the target images must be scattered among the GPUs in groups of rows. Hitmap provides a `hitDistribTile()` constructor to create tiles which are automatically distributed among the processors. This function receives the input native datatype, the shape of the global buffer to distribute, the name of a topology function that groups the processors to create neighbor relations, and the name of a layout defining the desired type of partition and distribution.

Since the Hitmap version of the algorithm presented in [11] was devoted to run in a single-node homogeneous multi-GPU cluster, the input images were sliced in equally distributed groups of rows. This is achieved by storing these images in distributed arrays according to a `Blocks` layout built on top of a topology that projects the processors along the rows dimension of the images (`ArrayDimProjection(HIT_ROWSDIM)`). The code snippet from Fig. 2a shows how these arrays are defined for both reference and target input images. The satisfactory results obtained in this previous work stem in part from the aforementioned GPU homogeneity. However, this kind of blocked equal distributions are very likely to lead to a quite uneven load balancing when run in heterogeneous clusters composed of host nodes and GPU devices with different computing capabilities and memory subsystems. A new weighted data partition has been introduced to balance the number of row-based slices sent to each GPU. Namely, it can be done by replacing the `Blocks` layout with a `DimWeighted_Copy` one, as shown in Fig. 2b. The constructor of this kind of layout needs to know the dimension along which the array will be distributed (in this case `HIT_ROWSDIM`, the rows dimension of the image) and a `HitWeights` object encapsulating the weights to apply. The code in Fig. 3 shows how to create such an object: first an array of weights must be declared, which is passed to a `hitWeights` constructor along with the number of processes. Ratios for data distribution are computed internally by dividing each weight by the sum of the array, taking into account the following two conditions: (1) if the weights array is longer than the number of processes, the excess values are discarded, (2) if it is shorter, then the array is completed with zeros and the corresponding processes will not receive data.

### 4.2. Preprocessing

This step is composed of three different parts, on which both reference and target images are first filtered using a Blackman window, then normalized, and finally shrunk to a reduced number of bands by means of a principal component analysis (PCA). In this distributed approach, the GPU commanded by each process takes the reference and target images slices loaded in Step 1 and filters them. Remember that these slices are stored in `HitTile` objects that were allocated in the host memory space of each process, so they must be previously copied to the global memory space of each GPU. Once filtered, they must be normalized by centering the value of each pixel with relation to the mean value of all the pixels of its band. Since each GPU only keeps its corresponding group of rows, an *all-reduce* operation is needed so that all the processes could have the full summation of each band of the input images, and then compute the mean values used to center their image slices. These filtering and centering stages are the first point on which the load balancing of the distributed arrays defined in Fig. 2b plays a role, since the higher the number of rows assigned to each process , the higher the requirements of the GPU in terms of memory and computing power.

The principal component analysis of a filtered and centered input image is composed of several stages. First, a correlation matrix of the input is calculated. This task involves a matrix product whose column size depends on the number of rows assigned to each process (thus, the load balancing of the distributed arrays also plays a role here) and an *all-reduce* operation

```
1 HitTile_float im1_tileRows =
    hitDistribTile( float, hitShape((
    BANDS),(ROWS),(COLUMNS)),
    ArrayDimProjection(HIT_ROWSDIM),
    Blocks );
2
3
4 HitTile_float im2_tileRows =
    hitDistribTile( float, hitShape((
    BANDS2),(ROWS2),(COLUMNS2)),
    ArrayDimProjection(HIT_ROWSDIM),
    Blocks );
```

(a) Fixed `Blocks` layout

```
1 HitTile_float im1_tileRows =
    hitDistribTile( float, hitShape((
    BANDS),(ROWS),(COLUMNS)),
    ArrayDimProjection(HIT_ROWSDIM),
    DimWeighted_Copy(HIT_ROWSDIM,
    im_tileRows_hitweights) );
2 HitTile_float im2_tileRows =
    hitDistribTile( float, hitShape((
    BANDS2),(ROWS2),(COLUMNS2)),
    ArrayDimProjection(HIT_ROWSDIM),
    DimWeighted_Copy(HIT_ROWSDIM,
    im_tileRows_hitweights) );
```

(b) Weighted `DimWeighted_Copy` layout

**Fig. 2.** Definition of row-distributed tiles in Hitmap for input images.

```
1 float im_tileRows_weights_array = {...}
2 HitWeights im_tileRows_hitweights = hitWeights(world_size,
    im_tileRows_weights_array);
```

**Fig. 3.** Example of Hitmap weight arrays definitions. Weight arrays must have as many elements as processes in the corresponding topology. Ratios for data distribution are computed by dividing each weight by the sum of the full weights array.

used to accumulate the partial results of that product and then broadcast the matrix to all the processes. More details of how this matrix is obtained in parallel are given in [12]. Second, each GPU uses cuSOLVER [3] to compute a private copy of the singular value decomposition (SVD) of the full correlation matrix, one of the matrices obtained from this decomposition being a transformation operator used by the GPU to get a band-reduced and principal-component ordered version of its row-sliced input. Each process retrieves the result of that transformation from its GPU and stores it for each image in the distributed tiles created by the code from Fig. 4. The same `im_tileRows_hitweights` object from Fig. 3 must be used to define the tiles for the row-sliced PCA results, as they are transformations of the corresponding weighted row-sliced inputs with the same number of rows. So this is the third point of this step at which the row-based load balancing is relevant. According to the algorithm, the result is restricted to a small number of bands given by a parameter named `PCAS`. In order to keep just the first `PCAS` bands, an additional `HitShpView` transformation is needed. The last part of the PCA calculation consists of a redistribution of the band-reduced input images, sketched in Fig. 5. By now, each process $k$ has, for each image, a slice of $c$ columns $\times$ $r_k$ rows $\times$ $eb$ bands, with $eb$ = `PCAS` and $r_k$ being the number of rows computed for the process $k$ from the `im_tileRows_hitweights` object. However, each GPU is expecting as inputs for Step 3, a group of $eb_k$ full bands, each one being composed of $r \times c$ elements. This requirement is fulfilled by rearranging the row-sliced PCA results into band-sliced distributed arrays. Fig. 6 shows how these band-sliced arrays are created. In this case, the processors of the underlying topology are projected along the bands dimension of the images (`ArrayDimProjection(HIT_BANDDIM)`), and the `DimWeighted_Copy` layout includes a different `HitWeights` object (`im_tileBands_hitweights`) that will set how many bands will be received by each process. The rearrangement operation is performed by means of the `hit_patternDoRedistribute()` function of Hitmap, which implements a transparent mechanism to rearrange the data of a distributed `HitTile` into another tile with a different mapping and only expects the input and output tiles as arguments.

### 4.3. Band processing and composition

In this step, each GPU loops over its $eb_k$ pairs of reduced bands of $c$ columns and $r$ rows, performing a high-pass filtering, a multilayer fractional Fourier transform (MLFFT) [27] and a log-polar coordinate transformation in both reference and target bands on each pair, which are then merged into a phase-correlated map (Stages II to IV in Fig. 1). Some of these operations are computed by cuFFT [2] routines. At the end of the iteration, the map is accumulated into an ancillary buffer, so the buffer will contain the partial log-polar map corresponding to the $eb_k$ reduced bands computed by the GPU at the end of the loop. Here lies the relevance in terms of computing demands of the load balancing of the distributed arrays defined in Fig. 6. Finally, in Stage V, the sum of all these partial maps is reduced to the master process (*leader* in Hitmap terminology) of the algorithm.

### 4.4. Peak processing

In the first part of this step, the *leader* process uses Thrust [4] to generate a host-side ordered indexes vector from the GPU-stored final average map. Due to the data sizes involved, this is more efficient than issuing a distributed sort. Both arrays are broadcast to the rest of the processes along with the first PCA component from both reference and target images, since all these data structures are needed to process the peaks pointed by a number of the top elements of the ordered indexes vector. Then, every process cyclically traverses the top subset of the ordered indexes array to obtain a partial maximum peak. The information from these partial peaks is on structures that are packed as custom Hitmap datatypes and then gathered in the leader process. Finally, this process inspects the partial peaks received and computes the expected outputs of the rotation *angle*, *scale* factor and cartesian *shift* shown in Fig. 1. The description of this step is included to make a comprehensive coverage of the algorithm, although all these operations are performed only in the leader process and all the weight-distributed arrays have been previously released.

```
1 HitTile_float ht_im1_pca_geosliced = hitDistribTile( float, hitShape( (BANDS),
      (ROWS), (COLUMNS) ), ArrayDimProjection(HIT_ROWSDIM), DimWeighted_Copy(
    HIT_ROWSDIM, im_tileRows_hitweights), hitShpView( (HIT_BANDDIM,
    HIT_SHAPE_FIRST, PCAS) ) );
2 HitTile_float ht_im2_pca_geosliced = hitDistribTile( float, hitShape( (BANDS2)
    , (ROWS2), (COLUMNS2) ), ArrayDimProjection(HIT_ROWSDIM), DimWeighted_Copy
    (HIT_ROWSDIM, im_tileRows_hitweights), hitShpView( (HIT_BANDDIM,
    HIT_SHAPE_FIRST, PCAS) ) );
```

**Fig. 4.** Definition of weight-distributed tiles in Hitmap for row-sliced PCA results.

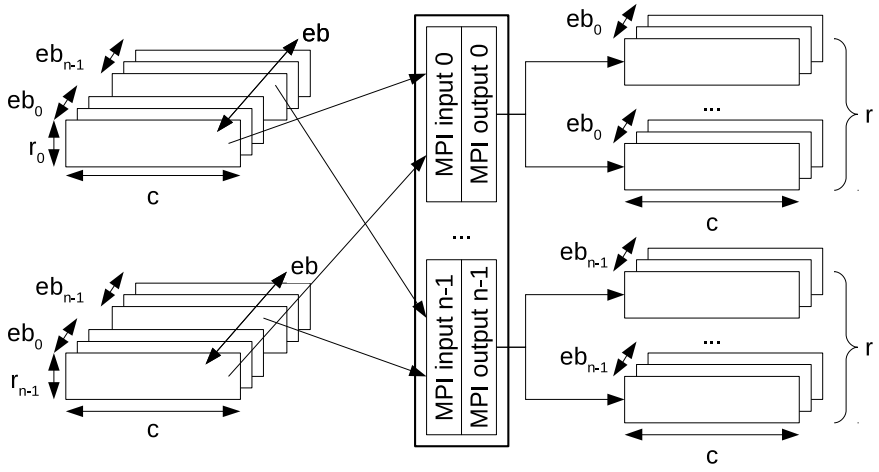


**Fig. 5.** Workflow of rows-to-bands group redistribution among $n$ processes. Row-distributed slices (left) have $c$ columns $\times$ $r_k$ rows $\times$ $eb$ bands. Band-distributed slices (right) have $c$ columns $\times$ $r$ rows $\times$ $eb_k$ bands.

```
1 HitTile_float im1_tileBands = hitDistribTile( float, hitShape( (PCAS), (ROWS),
      (COLUMNS) ), ArrayDimProjection(HIT_BANDDIM), DimWeighted_Copy(
    HIT_BANDDIM, im_tileBands_hitweights) );
2 HitTile_float im2_tileBands = hitDistribTile( float, hitShape( (PCAS), (ROWS2)
    , (COLUMNS2) ), ArrayDimProjection(HIT_BANDDIM), DimWeighted_Copy(
    HIT_BANDDIM, im_tileBands_hitweights) );
```

**Fig. 6.** Definition of weight-distributed tiles in Hitmap for band-sliced PCA results after rows-to-bands redistribution.

## 5. Experimental study

Single-node multi-GPU experiments described in [11] proved how a Hitmap-based implementation of the HYFM algorithm improved both the programmability and the performance of a previous MPI+CUDA version. In this section, we extend those results by introducing a distributed multi-GPU experimental study on which several pairs of hyperspectral images are registered, distributing the work using three different distribution policies in a heterogeneous cluster with two nodes that host five NVIDIA GPUs of three different generations and capabilities. Table 1 summarizes the main properties of the heterogeneous GPU cluster used for this experimental study. Four GPUs are two Tesla K40c, and two Titan Black GPU devices of Compute Capability 3.5, connected to a node (named *hydra*, also referred to as *H1*) powered by an Intel Xeon 2014Q3 E5-2609v3 processor. K40c GPUs have 12 Gb of global memory, while Titan Black GPUs have only 6 Gb of global memory. The last GPU is a high-end Tesla V100 of Compute Capability 7.0 hosted by a server with two Intel Xeon 2017Q3 Platinum 8160 processors (node named *manticore*, also referred as *H2*) with a total of 48 cores and 96 hardware threads. Both nodes are linked by a Gigabit Ethernet network fabric technology.

Regarding the load balancing scenarios, three work distributions have been defined by means of the `DimWeighted_Copy` layout: (a) *blocks*, that mimics the equal distribution of the `Blocks` layout used in [11], (b) *memory*, whose weights are set according to the global memory available in each GPU, and (c) *timeperf*, that tries to adapt the work distribution to the real computing capabilities of each GPU. Table 2 shows the weights set for both row-sliced and band-sliced distributed tiles in each load balancing scenario. For the *blocks* scenario, all weights are set to the same value (e.g. 1.0) for all the GPUs in both groups of distributed arrays in order to mimic the behavior of the original `Blocks` layout. For the *memory* scenario, the weights used to distribute the input arrays and the row-sliced PCA result arrays match the global memory size of each GPU in GB, as shown in Table 1. These weights are approximated with a configuration of integer numbers (1, 1, 1, 1, 4), whose sum matches the value of the PCAS parameter that determines the number of bands in the band-sliced PCA results array. The $PCAS = 8$ value was experimentally determined in [26]. For the *timeperf* scenario, the weights used to distribute the input arrays and the row-sliced PCA result arrays match a normalized performance metric $\rho_i = T/t_i$, where $i$ represents each different GPU model. We

**Table 1**
Main properties of the testbed used for the experimental study.

| | *hydra* (H1) | *manticore* (H2) |
|---|---|---|
| CPU (host) | 2× Intel Xeon E5-2609v3<br>1.9 GHz, 6 cores each<br>64 GB of RAM | 2× Intel Xeon Platinum 8160<br>2.1 GHz, 48 cores each<br>256 GB of RAM |
| GPU (devices) | 2×NVIDIA GTX TITAN Black<br>GK110B architecture (CC 3.5)<br>2880 CUDA cores, 6 GB RAM<br>2×NVIDIA Tesla K40c<br>GK180 architecture (CC 3.5)<br>2880 CUDA cores, 12 GB RAM | 1×NVIDIA Tesla V100<br>GV100 architecture (CC 7.0)<br>5120 CUDA cores, 32 GB RAM |
| CUDA environment | `nvcc` and device-side libraries from CUDA Toolkit 10.0CUDA driver 410.48 | |
| MPI implementation | mpich-3.2.1 | |
| Network fabric | Gigabit Ethernet | |

**Table 2**
Weights arrays for *blocks*, *memory* and *timeperf* load balancing layouts. H1 and H2 are the two nodes that make up the cluster.

| Load balancing | Weighted distribution | H1 | | | | H2 |
|---|---|---|---|---|---|---|
| | | TB | TB | K40 | K40 | V100 |
| *blocks* | *Row-based* | 1 | 1 | 1 | 1 | 1 |
| | *Band-based* | 1 | 1 | 1 | 1 | 1 |
| *memory* | *Row-based* | 6 | 6 | 12 | 12 | 32 |
| | *Band-based* | 1 | 1 | 1 | 1 | 4 |
| *timeperf* | *Row-based* | 0.25 | 0.25 | 0.20 | 0.20 | 1 |
| | *Band-based* | 1 | 1 | 1 | 1 | 4 |

**Table 3**
Normalized performance metrics used to set the *timeperf* weights array for the row-sliced distributed tiles. TB stands for NVIDIA GTX TITAN Black, K40 stands for NVIDIA Tesla K40c, and V100 stands for NVIDIA Tesla V100.

| GPU | Single-device wall time ($t_i$) | Baseline time ($T = min(t_i)$) | Normalized perf. ($\rho_i = T/t_i$) |
|---|---|---|---|
| TB | 16.15 s | – | $3.87/16.15 \sim 0.25$ |
| K40 | 20.79 s | – | $3.87/20.79 \sim 0.20$ |
| V100 | 3.87 s | 3.87 s | $3.87/3.87 = 1.00$ |

experimentally measure $t_i$ by executing the whole algorithm in a single GPU of model $i$, with a pair of small reference images with sizes 512 × 4096 × 224. This is the size of the largest test case run for a single GPU in [12]. Then, we normalize these times with the fastest performance time $T = min(t_i)$. In our experimental platform, this is the execution time obtained using the Tesla V100 GPU. Table 3 summarizes how these weights have been obtained. The weights for the band-sliced PCA results array, with a small number of bands ($PCAS = 8$), are approximated as in the *memory* scenario.

Table 4 contains, for each pair of *cols* × *rows* × *bands* input images, the average ($\bar{x}$) and the standard deviation ($\sigma$) of the wall time after 10 runs of the algorithm, distributing the work among the five GPUs available according to the *blocks*, *memory* and *timeperf* load balancing layouts. The HYFM algorithm properties allow the use of random inputs for performance evaluation purposes, as its performance is non-sensitive to translation, rotation, scale, and noise in the input images [29]. Thus, all the tests have been run using randomly generated matrices as synthetic hyperspectral images. We have chosen a number of 224 bands, since it can be considered a representative value for very high spectral resolution images [1]. For all the test cases, we choose a number of 4096 rows to be distributed among GPUs. It is the number of rows in the largest images included in [11], which have a size of 1024 × 4096 × 224. The number of columns chosen ranges from 1024 to 4096 in increments of 1024, to create increasing workloads to be distributed. We now compare the three distribution policies or layouts in terms of their load balancing, stability and GPU usage properties. The GPU usage is measured as the sum of GPU kernel times and host-to-device, device-to-host and device-to-device transfers, obtained by means of the `nvprof` CUDA profiler, for each of the five GPUs available in the cluster. Table 5 shows the GPU time for each device, along with the corresponding average time and its standard deviation. These times were obtained for a pair of random images of size 4092 × 4096 × 224. It is the largest input size that does not exhaust the global memory of any of the 5 GPUs, for any load balancing layout, when running the code instrumented with the *nvprof* tools. For completeness, this additional test case was also run with no profiling activity and was hence included as a separate row in Table 4. Both the *blocks* and the *memory* layouts show a relevant load unbalance. When *blocks* is applied, all the GPUs must perform the same amount of work, which leads the high-end V100 to clearly outperform the rest of the devices, whose capabilities are much more modest and are thus overloaded. The V100 GPU is idle and waiting for the others at each synchronization point. A standard deviation of 4.54 (43.61% over the average GPU time) proves those differences. Regarding the *memory* layout, it increases the amount of work performed by the V100 and thus alleviates the load for the other GPUs (a V100 has 2.6× more memory than a K40 and 5.2× more than a TB). Thus, the average GPU time decreases from 10.41 s for *blocks* to 8.39 s. However, it is not enough to compensate the computing capabilities of the V100, as shown by the high standard deviation of 3.33 (39.65% over the average GPU time). Nevertheless, such a redistribution could be useful to compute inputs that are too large to be processed using the *blocks* layout, but can be fitted in the global memory of all the GPUs. By contrast, the GPU times achieved when applying the *timeperf* layout prove that it really does provide a quite balanced work distribution. The average GPU time has fallen to 5.59 s and the standard deviation becomes almost negligible (0.35, barely 6.25% of the average). Thus, trying to adapt the work distribution to the computing capabilities of each GPU seems to be a good technique for the distributed multi-GPU implementation of the HYFM algorithm. As the detailed descriptions included in [11,12] show, the distributed implementation of the algorithm requires many synchronization points and data transfers among processes. A balanced GPU work distribution contributes to a smooth progress of the execution, as it is expected to minimize the wait times of processes at such points.

The results obtained for all the test cases show the trend that the bigger the inputs, the larger the impact that the configuration of a proper load balancing layout has in performance terms, for the distributed multi-GPU implementation of the HYFM algorithm. As expected, the *timeperf* layout, which was intended to adapt the work distribution to the computing capabilities of each GPU, arises in all the test cases as the fastest option.

**Table 4**
Performance results (mean wall time $\bar{x}$ and standard deviation $\sigma$, with $N = 10$ runs for each test case) for *blocks*, *memory* and *timeperf* load balancing layouts. These times include CPU processing, network communication costs, and device times. Device times are further analyzed in Table 5.

| Input images size (*cols* × *rows* × *bands*) | $N = 10$ runs each test case | | | | | |
|---|---|---|---|---|---|---|
| | *blocks* | | *memory* | | *timeperf* | |
| | $\bar{x}$ | $\sigma$ | $\bar{x}$ | $\sigma$ | $\bar{x}$ | $\sigma$ |
| 1024 × 4096 × 224 | 12.81 s | 0.70 | 12.18 s | 0.49 | 11.94 s | 0.32 |
| 2048 × 4096 × 224 | 17.41 s | 1.28 | 15.77 s | 0.33 | 15.17 s | 0.57 |
| 3072 × 4096 × 224 | 20.01 s | 0.76 | 19.50 s | 0.25 | 19.20 s | 0.88 |
| 4096 × 4096 × 224 | 27.48 s | 1.96 | 27.68 s | 0.72 | 23.45 s | 0.88 |
| 4092 × 4096 × 224 | 26.06 s | 1.47 | 24.25 s | 0.29 | 19.26 s | 0.41 |

**Table 5**
Comparison of load balance achieved by *blocks*, *memory* and *timeperf* layouts for inputs of size 4092 × 4096 × 224. Each row shows the device time (sum of memory transfers and computing) consumed by each GPU for a layout, along with the corresponding mean $\bar{x}$ and standard deviation, both in absolute terms ($\sigma$) and in percentage (%) of $\bar{x}$.

| Load balancing layout | H1 | | | | H2 | $\bar{x}$ | $\sigma$ (%) |
|---|---|---|---|---|---|---|---|
| | TB | TB | K40 | K40 | V100 | | |
| *blocks* | 11.67 s | 13.21 s | 11.85 s | 12.92 s | 2.38 s | 10.41 s | 4.54 (43.61%) |
| *memory* | 7.52 s | 5.60 s | 10.96 s | 12.72 s | 5.16 s | 8.39 s | 3.33 (39.65%) |
| *timeperf* | 5.83 s | 5.99 s | 5.11 s | 5.39 s | 5.62 s | 5.59 s | 0.35 (6.25%) |

## 6. Conclusions

In this paper, we present an extension of the distributed implementation of the HYFM hyperspectral image registration algorithm from [11,12]. The main feature added to this extended version consists of a load balancing mechanism to make the code able to adapt its performance to the capabilities of a heterogeneous multi-GPU cluster. We review the main functionalities offered by the Hitmap library used in the implementation to manage distributed arrays, including the new ones that support weighted distributions of data among processes. Then, we recall how the work of each step of the algorithm is divided among several GPUs, explaining how the new weight-distributed arrays of Hitmap are used to support the aforementioned load balancing mechanism. An experimental study to assess the impact that a proper load balancing has on the performance of the distributed version of the code has been conducted. This study consists of registering pairs of random hyperspectral images with sizes up to 4096 × 4096 × 224 on a heterogeneous cluster composed by two nodes with different Intel Xeon CPU families and hosting five NVIDIA GPUs of different capabilities and generations. The workload is distributed following three different data layouts: (a) *blocks*, that applies a blocked equal distribution, (b) *memory*, that proportionally assigns more work to those GPUs equipped with more global memory, and (c) *timeperf*, that tries to distribute the workload according to the computing capabilities of each device. The results show that the bigger the inputs, the larger the performance impact of a proper load balancing layout has in this code. A comparison of these three layouts was also proposed to check to what extent each one is really balanced or not in terms of GPU usage. This comparison confirmed the *blocks* layout as quite unbalanced ($\sigma = 4.54$, 43.61% over the average GPU time). The *memory* one was revealed as insufficient to compensate the performance differences among GPUs ($\sigma = 3.33$, 39.65% over the average). By contrast, the *timeperf* layout indeed provides a load-balanced scenario among the 5 GPUs ($\sigma = 0.35$, 6.25% over the average). The adaptation of the work distribution to the computing capabilities of each GPU therefore seems to be a good load balancing technique for the distributed multi-GPU implementation of the HYFM algorithm. A balanced GPU work distribution contributes to a smooth progress of the execution, as it is expected to minimize the wait times of processes at synchronization points and in collective data transfers. For all the test cases, the *timeperf* layout obtained the best execution wall time.

Finally, some future research lines are proposed. Improvements in the distributed implementation of some parts of the algorithm, such as the principal component analysis (PCA) or the band processing stages based on cuFFT operations, will reduce the number of synchronization points and data transfer operations among processes, which is expected to increase the relevance that a well-balanced GPU usage has in the algorithm wall time. Additionally, the integration of Hitmap with the Controllers model [23] for generic accelerator programming could also simplify the programming and portability of the GPU code.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgments

We would like to thank Dr. Francisco Argüello, Dr. Dora B. Heras and their team at Universidade de Santiago de Compostela, Spain, for sharing with us the tools and knowledge needed to start this research. This work has been funded by the Consejería de Educación of Junta de Castilla y León and the European Regional Development Fund (ERDF) program (projects PROPHET, VA082P17, and PROPHET-2, VA226P20); by the Ministerio de Economía, Industria y Competitividad of Spain (project PCAS, TIN2017-88614-R); and by the Fulbright Commission, (grant Salvador de Madariaga/Fulbright Scholar PRX17/00674).

## References

[1] NASA Jet Propulsion Laboratory, Airborne visible/infrared imaging spectrometer (AVIRIS) database, 2019, URL: https://aviris.jpl.nasa.gov/data/index.html, Last access: 2019-12-04.
[2] NVIDIA Corporation, CuFFT library user's guide, 2019, URL: https://docs.nvidia.com/cuda/pdf/CUFFT_Library.pdf, Last access: 2019-12-04.
[3] NVIDIA Corporation, CuSOLVER library user's guide, 2019, URL: https://docs.nvidia.com/cuda/pdf/CUSOLVER_Library.pdf, Last access: 2019-12-04.
[4] NVIDIA Corporation, Thrust quick start guide, 2019, URL: https://docs.nvidia.com/cuda/pdf/Thrust_Quick_Start_Guide.pdf, Last access: 2019-12-04.

[5] F. Alonso, A. Gonzalez-Escribano, Y. Torres, D.R. Llanos, Mecanismo de equilibrado de carga en sistemas heterogéneos, in: Avances En Arquitectura y Tecnología de Computadores. Actas de Las Jornadas SARTECO 2019, 2019, pp. 294–300.
[6] C.A. Bohn, G.B. Lamont, Load balancing for heterogeneous clusters of PCs, Future Gener. Comput. Syst. 18 (3) (2002) 389–400, http://dx.doi.org/10.1016/S0167-739X(01)00058-9.
[7] F. Bonelli, M. Tuttafesta, G. Colonna, L. Cutrone, G. Pascazio, An MPI-CUDA approach for hypersonic flows with detailed state-to-state air kinetics using a GPU cluster, Comput. Phys. Comm. 219 (2017) 178–195, http://dx.doi.org/10.1016/j.cpc.2017.05.019.
[8] B. Chamberlain, D. Callahan, H. Zima, Parallel programmability and the Chapel language, Int. J. High Perform. Comput. Appl. 21 (3) (2007) 291–312, http://dx.doi.org/10.1177/1094342007078442.
[9] Y. Chen, X. Cui, H. Mei, PARRAY: A unifying array representation for heterogeneous parallelism, SIGPLAN Not. 47 (8) (2012) 171–180, http://dx.doi.org/10.1145/2370036.2145838.
[10] J. Drönner, N. Korfhage, S. Egli, M. Mühling, B. Thies, J. Bendix, B. Freisleben, B. Seeger, Fast cloud segmentation using convolutional neural networks, Remote Sens. 10 (11) (2018) http://dx.doi.org/10.3390/rs10111782, paper 1782.
[11] J. Fernández-Fabeiro, A. Gonzalez-Escribano, D.R. Llanos, Simplifying the multi-GPU programming of a hyperspectral image registration algorithm, in: International Conference on High Performance Computing Simulation (HPCS), Dublin, Ireland, 2019, pp. 11–18, http://dx.doi.org/10.1109/HPCS48598.2019.9188064.
[12] J. Fernández-Fabeiro, Á. Ordóñez, A. Gonzalez-Escribano, D.B. Heras, A multi-device version of the HYFMGPU algorithm for hyperspectral scenes registration, J. Supercomput. 75 (3) (2019) 1551–1564, http://dx.doi.org/10.1007/s11227-018-2689-7.
[13] B.B. Fraguela, G. Bikshandi, J. Guo, M.J. Garzarán, D. Padua, C. von Praun, Optimization techniques for efficient HTA programs, Parallel Comput. 38 (9) (2012) 465–484, http://dx.doi.org/10.1016/j.parco.2012.05.002.
[14] J. Fresno, A. Gonzalez-Escribano, D.R. Llanos, Blending extensibility and performance in dense and sparse parallel data management, IEEE Trans. Parallel Distrib. Syst. 25 (10) (2014) 2509–2519, http://dx.doi.org/10.1109/TPDS.2013.248.
[15] J. Gao, Y. Sun, B. Zhang, Z. Chen, L. Gao, W. Zhang, Multi-GPU based parallel design of the ant colony optimization algorithm for endmember extraction from hyperspectral images, Sensors 19 (3) (2019) http://dx.doi.org/10.3390/s19030598, paper 598.
[16] A.S. Garea, D.B. Heras, F. Argüello, Caffe CNN-based classification of hyperspectral images on GPU, J. Supercomput. 75 (3) (2019) 1065–1077, http://dx.doi.org/10.1007/s11227-018-2300-2.
[17] A. Gonzalez-Escribano, Y. Torres, J. Fresno, D.R. Llanos, An extensible system for multilevel automatic data partition and mapping, IEEE Trans. Parallel Distrib. Syst. 25 (5) (2014) 1145–1154, http://dx.doi.org/10.1109/TPDS.2013.83.
[18] H. Jiang, S. Chen, D. Li, C. Wang, J. Yang, Papaya tree detection with UAV images using a GPU-accelerated scale-space filtering method, Remote Sens. 9 (7) (2017) http://dx.doi.org/10.3390/rs9070721, paper 721.
[19] L. Linghu, J. Wu, Z. Wu, X. Wang, Parallel computation of EM backscattering from large three-dimensional sea surface with CUDA, Sensors 18 (11) (2018) http://dx.doi.org/10.3390/s18113656, paper 3656.
[20] J. López-Fandiño, D. B. Heras, F. Argüello, M. Dalla Mura, GPU framework for change detection in multitemporal hyperspectral images, Int. J. Parallel Program. 47 (2) (2019) 272–292, http://dx.doi.org/10.1007/s10766-017-0547-5.
[21] D.A. Mallón, A. Gómez, J.C. Mouriño, G.L. Taboada, C. Teijeiro, J. Touriño, B.B. Fraguela, R. Doallo, B. Wibecan, UPC performance evaluation on a multicore system, in: Proceedings of the Third Conference on Partitioned Global Address Space Programing Models, in: PGAS '09, ACM, New York, NY, USA, 2009, pp. 9:1–9:7, http://dx.doi.org/10.1145/1809961.1809974.
[22] E. Martel, R. Lazcano, J. López, D. Madroñal, R. Salvador, S. López, E. Juarez, R. Guerra, C. Sanz, R. Sarmiento, Implementation of the principal component analysis onto high-performance computer facilities for hyperspectral dimensionality reduction: Results and comparisons, Remote Sens. 10 (6) (2018) http://dx.doi.org/10.3390/rs10060864, paper 864.
[23] A. Moreton-Fernandez, A. Gonzalez-Escribano, D.R. Llanos, Multi-device controllers: A library to simplify parallel heterogeneous programming, Int. J. Parallel Program. 47 (1) (2019) 94–113, http://dx.doi.org/10.1007/s10766-017-0542-x.
[24] A. Moreton-Fernandez, H. Ortega-Arranz, A. Gonzalez-Escribano, Controllers: An abstraction to ease the use of hardware accelerators, Int. J. High Perform. Comput. Appl. 32 (6) (2018) 838–853, http://dx.doi.org/10.1177/1094342017702962.
[25] A. Ordóñez, F. Argüello, D.B. Heras, Fourier–Mellin registration of two hyperspectral images, Int. J. Remote Sens. 38 (11) (2017) 3253–3273, http://dx.doi.org/10.1080/01431161.2017.1292071.
[26] A. Ordóñez, F. Argüello, D.B. Heras, GPU accelerated FFT-based registration of hyperspectral scenes, IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens. 10 (11) (2017) 4869–4878, http://dx.doi.org/10.1109/JSTARS.2017.2734052.
[27] W. Pan, K. Qin, Y. Chen, An adaptable-multilayer fractional fourier transform approach for image registration, IEEE Trans. Pattern Anal. Mach. Intell. 31 (3) (2009) 400–414.
[28] F. Rançon, L. Bombrun, B. Keresztes, C. Germain, Comparison of SIFT encoded and deep learning features for the classification and detection of Esca Disease in Bordeaux Vineyards, Remote Sens. 11 (1) (2018) http://dx.doi.org/10.3390/rs11010001, paper 1.
[29] B.S. Reddy, B.N. Chatterji, An FFT-based technique for translation, rotation, and scale-invariant image registration, IEEE Trans. Image Process. 5 (8) (1996) 1266–1271, http://dx.doi.org/10.1109/83.506761.
[30] E. Rodriguez-Gutiez, F. Martinez-Gil, J.M. Orduña, A. Gonzalez-Escribano, MARL-Ped+Hitmap: Towards improving agent-based simulations with distributed arrays, in: Algorithms and Architectures for Parallel Processing, Springer International Publishing, 2016, pp. 212–225, http://dx.doi.org/10.1007/978-3-319-49956-7_17.
[31] M. Rossi, P. Trslić, S. Sivčev, J. Riordan, D. Toal, G. Dooly, Real-time underwater stereofusion, Sensors 18 (11) (2018) http://dx.doi.org/10.3390/s18113936, paper 3936.
[32] B. Yang, M. Yang, A. Plaza, L. Gao, B. Zhang, Dual-mode FPGA implementation of target and anomaly detection algorithms for real-time hyperspectral imaging, IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens. 8 (6) (2015) 2950–2961, http://dx.doi.org/10.1109/JSTARS.2015.2388797.
[33] W. Zhang, G. Luo, L. Shen, T. Page, P. Li, M. Jiang, P. Maass, J. Cong, FPGA acceleration by asynchronous parallelization for simultaneous image reconstruction and segmentation based on the mumford-shah regularization, in: P.J. Bones, M.A. Fiddy, R.P. Millane (Eds.), Image Reconstruction from Incomplete Data VIII, Vol. 9600, International Society for Optics and Photonics, 2015, pp. 106–117, http://dx.doi.org/10.1117/12.2187898.
[34] S. Zhuang, P. Wang, B. Jiang, G. Wang, C. Wang, A single shot framework with multi-scale feature fusion for geospatial object detection, Remote Sens. 11 (5) (2019) http://dx.doi.org/10.3390/rs11050594, paper 594.

**Jorge Fernández-Fabeiro** received the M.S. degree in High Performance Computing and the Ph.D. degree in Research in Information Technologies from the Universidade da Coruña, Spain, in 2012 and 2017, respectively. His research activity during that period was focused on the achievement of automatic performance portability on heterogeneous environments. From March 2018 to October 2019 he was with the University of Valladolid as a postdoctoral researcher for the PROPHET project, working on the parallelization of sequential algorithms for scientific and engineering applications and its deployment in heterogeneous interconnected systems. He also collaborated as a teaching assistant for both universities in undergraduate-level computer architecture and parallel computing courses.

**Arturo Gonzalez-Escribano** received his MS and Ph.D. degrees in Computer Science from the University of Valladolid, Spain, in 1996 and 2003, respectively. Dr. Gonzalez-Escribano is Associate Professor of Computer Science at the Universidad de Valladolid since 2008. His research interests include parallel and distributed computing, parallel programming models, portability in heterogeneous systems, and embedded computing. More information about his current research activities can be found at http://www.infor.uva.es/~arturo.

**Diego R. Llanos** received his MS and Ph.D. degrees in Computer Science from the University of Valladolid, Spain, in 1996 and 2000, respectively. He is a recipient of the Spanish government's national award for academic excellence. Prof. Llanos is Full Professor of Computer Architecture at the Universidad de Valladolid, and his research interests include parallel and distributed computing, automatic parallelization of sequential code, and embedded computing. He is a Senior Member of the IEEE and Senior Member of the ACM, and has co-founded RDNest, a company that transfers to market research results in the field of Internet of Things and highperformance computing. More information about his current research activities can be found at http://www.infor.uva.es/~diego.